Report Number: WUCSE-2003-26

2003-04-24

Secure Sharing of Tuple Spaces in Ad Hoc Settings

Radu Handorean and Gruia-Catalin Roman

Security is emerging as a growing concern throughout the distributed computing community. Typical solutions entail specialized infrastructure support for authentication, encryption and access control. Mobile applications executing over ad hoc wireless networks present designers with a rather distinct set of security requirements. A totally open setting and limited resources call for lightweight and highly decentralized security solutions. In this paper we propose an approach that relies on extending an existing coordination middleware for mobility (Lime). The need to continue to offer a very simple model of coordination that assures rapid software development led to limiting extensions solely to password... **Read complete abstract on page 2.**

Secure Sharing of Tuple Spaces in Ad Hoc Settings

Radu Handorean and Gruia-Catalin Roman

Complete Abstract:

Security is emerging as a growing concern throughout the distributed computing community. Typical solutions entail specialized infrastructure support for authentication, encryption and access control. Mobile applications executing over ad hoc wireless networks present designers with a rather distinct set of security requirements. A totally open setting and limited resources call for lightweight and highly decentralized security solutions. In this paper we propose an approach that relies on extending an existing coordination middleware for mobility (Lime). The need to continue to offer a very simple model of coordination that assures rapid software development led to limiting extensions solely to password protected tuple spaces and per tuple access control. Password distribution and security are relegated to the application realm. Host level security is ensured by the middleware design and relies on standard support provided by the Java system. Secure interactions among agents across hosts are accomplished by careful exploitation of the interceptor pattern and the use of standard encryption. The paper explains the design strategy used to add security support in Lime and its implications for the development of mobile applications over ad hoc networks.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-26

Secure Sharing of Tuple Spaces in Ad Hoc Settings

Authors: Radu Handorean and Gruia-Catalin Roman

April 24, 2003

Abstract: Security is emerging as a growing concern throughout the distributed computing community. Typical solutions entail specialized
infrastructure support for authentication, encryption and access control. Mobile applications executing over ad hoc wireless
networks present designers with a rather distinct set of security requirements. A totally open setting and limited resources call for
lightweight and highly decentralized security solutions. In this paper we propose an approach that relies on extending an existing
coordination middleware for mobility (Lime). The need to continue to o®er a very simple model of coordination that assures
rapid software development led to limiting extensions solely to password protected tuple spaces and per tuple access control.
Password distribution and security are relegated to the application realm. Host level security is ensured by the middleware design
and relies on standard support provided by the Java system. Secure interactions among agents across hosts are accomplished by

Secure Sharing of Tuple Spaces in Ad Hoc Settings

Radu Handorean and Gruia-Catalin Roman
Mobile Computing Laboratory
Department of Computer Science and Engineering
Washington University
Saint Louis, Missouri 63130-4899, USA
{raduh, roman}@cse.wustl.edu
http://mobilab.cse.wustl.edu

Abstract

Security is emerging as a growing concern throughout the distributed computing community. Typical solutions entail specialized infrastructure support for authentication, encryption and access control. Mobile applications executing over ad hoc wireless networks present designers with a rather distinct set of security requirements. A totally open setting and limited resources call for lightweight and highly decentralized security solutions. In this paper we propose an approach that relies on extending an existing coordination middleware for mobility (LIME). The need to continue to offer a very simple model of coordination that assures rapid software development led to limiting extensions solely to password protected tuple spaces and per tuple access control. Password distribution and security are relegated to the application realm. Host level security is ensured by the middleware design and relies on standard support provided by the Java system. Secure interactions among agents across hosts are accomplished by careful exploitation of the interceptor pattern and the use of standard encryption. The paper explains the design strategy used to add security support in LIME and its implications for the development of mobile applications over ad hoc networks.

1 Introduction

Ad hoc networks are formed when hosts equipped with wireless communication capabilities interact with each other directly without support from any fixed wired infrastructure. Hosts can range greatly in both computational power and communication capabilities. Standard computers may be placed on mobile platforms (e.g., cars) and may be given continuous access to a reliable power source. Laptops and palmtops may be carried by individuals or small robots and subject to power limitations. Small processors may be embedded in specialized devices or integrated within miniature sensor systems. For the purpose of this paper, our interest is in applications that execute on computing devices that are sufficiently powerful to run Java software, are highly mobile, and do not rely in any way on the wired infrastructure. A world in which each individual carries a PDA but base stations are absent is a good metaphor for the setting we have in mind. Disaster response, mine exploration, low profile military action, social gatherings are representative application domains for our work. In all these cases network formation is opportunistic, its structure is subject to evolution, disconnections are a way of life, and the size of the community is constrained by the range of the wireless transmitters. Ad hoc routing, when available, may significantly expand the number of participating hosts.

Application development targeted to such open and dynamic settings is particularly difficult and coordination methods have been proposed as a possible software engineering solution. The basic idea is that of offering the developer a simple application-programming interface (API) that facilitates spatial and temporal decoupling among software components. In Linda [1], for instance, the API consists of a small set of operations that offer content-based access to tuples stored in a persistent global tuple space. Computation is relegated to local processing taking place in each component with the communication mechanics being completely hidden behind a high-level coordination model. The result is a significant reduction in the application development effort. LIME (Linda In a Mobile Environment) [2] is a coordination model and associated middleware that sought to extend this basic idea to mobility. In LIME, applications are constructed out of components called agents that represent the basic unit of modularity, execution and mobility. Agents reside on hosts and can move among them as long as connectivity is available. Agents residing

on hosts within communication range form a group. Group membership changes as communication links break down and get reestablished. Engagement and disengagement are the terms used to refer to joining and leaving a given group. An agent may create tuple spaces that can be shared with other agents within the same group. Each tuple space has a name and identically named tuple spaces belonging to agents in the same group are shared as if they were a single global tuple space. The latter is referred to as a federated tuple space. As groups change membership the content of each federated tuple space changes as well, with departing agents taking their tuples along and arriving agents contributing new tuples.

Ease of coordination within an open environment is a great asset but it must be tempered by security concerns. Many strategies commonly used in wired networks become problematic in the ad hoc setting. There is no protection against eavesdropping, there are no trusted authentication servers, there are no centralized databases of secure information, etc. Moreover, any proposed solution must be sensitive to resource utilization. Lightweight solutions are preferred but they must be able to work in settings where one cannot anticipate who will show up when and for how long. Full transparency may be desirable but the ability to hide security concerns from the application developer and user may not always be feasible. In this paper, we pose the question whether the coordination strategy made available in LIME can be made secure with minimal impact on the LIME middleware and on its fundamental coordination model. The goal is to maintain the rapid application development advantage of LIME while making available both secure and open access to the data shared through tuple spaces.

Our solution was to extend the LIME API in two important ways. First, we offer password-protected access to tuple spaces. The sharing policy within a group is extended to require not just the same name but protection with the same password. Within a single group, identically named tuple spaces are further partitioned according to their associated passwords. This is complemented by the ability to password-protect individual tuples regardless whether they are part of a protected or unprotected tuple space. Interestingly enough, the implementation of these two capabilities employs distinct features of the underlying LIME system. Moreover, by exploiting the fact that LIME restricts tuple space access to its creator agent, password usage is limited solely to tuple space creation, thus minimizing the scope of API modifications and also affording some level of robustness in regard to possible programming errors involving incorrect password utilization. In the final analysis, by making effective use of the existing LIME design the secure version of the system ends up to be a sandwiching of the existing middleware between a security veneer above and an interceptor below. The latter provides the proper encryption of messages associated with secure tuple spaces using a protected table shared with the former. The price we pay for achieving this level of simplicity is the need to accomplish the initial password distribution possibly outside of the application itself and the requirement for the application to manage required password changes in response to possible security compromises.

The remainder of the paper is structured as follows. Section 2 reviews the LIME coordination model. This is necessary since the design relies heavily on both technical features of LIME and on extending its semantics to security. Section 3 explains our security extensions to the original model. Section 4 presents the implementation strategy. Section 5 describes the test application we've developed to evaluate the security extensions we implemented. Related work on secure coordination is presented in Section 6. Conclusions appear in Section 7.

2 The Lime Coordination Model

Because this effort builds directly on LIME and exploits some of its more subtle technical features, we start our presentation with an overview the LIME model and illustrate it by means of a simple example involving a group of people who, while present at the same locale, communicate with each other via a chat program running on PDAs equipped with a wireless capability at the level of the 802.11b protocol.

The LIME middleware supports the development of applications exhibiting physical mobility of hosts and logical mobility of agents. An agent is a software component that may reside permanently on a host or may move from one host to another connected host. Hosts can move in physical space, serve as containers for the agents, and run local versions of the LIME system server. As suggested earlier, LIME extends the coordination model of Linda in significant ways. First, the globally accessed persistent tuple space of Linda is replaced in LIME by transient sharing of identically named tuple spaces belonging to agents that reside on hosts that are mutually accessible over the ad hoc network. Other LIME extensions to Linda include location specific operations, transparent tuple migration, and the ability to react to the presence or appearance of tuples within specific transiently shared tuple spaces.

Transparent Context Maintenance. The model underlying LIME accomplishes the shift from a fixed global

context to a dynamically changing one by distributing the single Linda tuple space across multiple tuple spaces, each local to an agent, and by introducing rules for transient sharing of the individual tuple spaces based on naming and connectivity; LIME allows an agent to structure its holdings across multiple tuple spaces each being shared only with other identically named tuple spaces local to other agents within the group. Group membership is controlled by connectivity among hosts. Sharing of multiple tuple spaces results in the formation of a virtual global data structure called a federated tuple space. The content of the federated tuple space is the union of the contents associated with the contributing tuple spaces. Access to the federated tuple space is accomplished by simply accessing the API for the local tuple space. After sharing, local actions have global effects. Simplicity is achieved by accessing solely local tuple spaces regardless of the network setting. Context awareness and coordination is achieved by transparent maintenance of a broader computational context and by transparent extension of the effects of what otherwise appear to be local actions. The agent's gateway to the federated tuple space is called the *interface tuple space* (ITS).

Basic access to the ITS takes place using the traditional Linda primitives (e.g., **in**, **rd**, **out**), whose semantics remain essentially unaffected. The **out** operation takes a tuple t and places it into a tuple space; **in** takes as parameter a template p and blocks until a tuple matching the template is written to the tuple space at which point **in** returns a copy of that tuple, after removing the original from the tuple space; **rd** exhibits a similar behavior but it leaves the original in the tuple space–the details of the matching mechanism will be explained later. LIME offers also non-blocking versions of **in** and **rd** in the form of probe variants of the same operations (e.g., **inp**, **rdp**). In general, non-blocking operations return a matching tuple (if one is available) and null otherwise. Both blocking and non-blocking extensions designed to handle entire groups of tuples matching the same template are also included in LIME.

A simple implementation of the chat program can be readily accomplished by having one agent per PDA with each agent initially creating a single shared tuple space called *"Chat_Room."* A message-sending request is transformed into placing in the tuple space a tuple containing the user id, the user name, a sequence number, and the message text. All other agents in the group gain access to the newly generated tuple by issuing a **rd** operation with an appropriate template on their own local tuple space with the same name. The originator of the message can remove it at some later point by employing a removal policy based on time to live. An alternate approach might be to implement a simple logical clock protocol through proper manipulation of the sequence numbers. In all cases the size of the resulting chat program is very small.

As in Linda, a tuple consists of an ordered list of fields. Each field has a type and a value. A template is an ordered list of fields that can contain type designators (*formal* fields) or explicit values (*actual* fields). A tuple and a template are said to match if both contain the same number of fields and each corresponding pair of fields matches. LIME was extended with the ability to specify the matching policy on a field by field basis. The field-level matching policies available are: (1) Exact type matching allows the field in the template to be a formal but requires its type to be the same as the type of the object in the corresponding tuple field. (2) Exact value match asks the template field to provide an actual that will match exactly the type and the value of the corresponding field in the tuple.

Controlling Context Awareness. A read-only tuple space called the `LimeSystemTupleSpace` provides an agent with a view of the overall system configuration. Its tuples contain information about the mobile agents present in the community, physical hosts they execute on, and tuple spaces created for coordination. Standard tuple space operations on `LimeSystemTupleSpace` allow an agent to respond to the arrival and departure of other agents and hosts. If we make the simplifying assumption that all the agents in the group are part of the chat room, an agent can easily build a list of who is around by examining `LimeSystemTupleSpace`.

Furthermore, LIME provides fine-grained control over the context on which an agent chooses to operate by extending its operations with tuple location parameters that define projections of the federated tuple space. LIME expresses tuple location parameters in terms of agent identifiers and host identifiers. These identifiers can be used to place tuples at a particular agent location or to restrict queries to specific agents or hosts.

Reacting to Changes in Context. Mobility entails a highly dynamic environment, where reacting to changes constitutes a major fraction of the application design. Therefore, LIME extends the basic Linda tuple space with the notion of *reaction*. A reaction $\mathcal{R}(s, p)$ is defined by a code fragment s that specifies the actions to be executed when a tuple matching the template p is found in the tuple space. After each operation on the tuple space, LIME non-deterministically selects a reaction and compares the template p against the tuple space contents. If a matching tuple is found, s is executed, otherwise the reaction is a skip. This selection and execution proceeds until there are no reactions enabled, and normal processing resumes. Thus, reactions are executed as if they belonged to a separate reactive program which runs to fixed point after each non-reactive statement. Blocking operations are not allowed in

s, as they could prevent the program from reaching fixed point.

This idealized perspective of reactions semantics is tempered in LIME by the pragmatics of an effective implementation. As such, reactions in LIME come in two forms: *strong reactions* and *weak reactions*. Strong reactions execute atomically with the writing of the tuple that enables them. These reactions are not allowed over the entire federated tuple space; they must always be restricted to a host or agent. Otherwise, maintaining the requirements of atomicity and serialization imposed by strong reactive statements would require a distributed transaction encompassing multiple hosts for every tuple space operation. LIME also provides the notion of *weak reaction*. The processing of a weak reaction proceeds as in the case of strong reactions, except that the execution of s does not happen atomically with the detection of a tuple matching p; instead, it is guaranteed to take place eventually if connectivity is preserved. This eliminates the need for a distributed transaction and allows this type of reaction to be installed and to execute over the federated tuple space.

Our earlier solution for the chat room can be simplified greatly through the use of reactions. Each agent in the chat room can register a weak reaction for tuples containing messages in the chat room. By doing so, when a tuple is inserted in the tuple space all reactions fire initiating eventual delivery of a copy of the message to the respective agents. Only after all these reactions are completed, local processing can resume on the PDA that sent the message. By now, it is known that the message delivery has been initiated already and the agent can remove it safely by issuing an **in** operation prior to continuing its local processing. The resulting code assumes the following general structure.

```
LimeTupleSpace lts = new LimeTupleSpace("Chat_Room");
lts.addWeakReaction(messageTemplate, reactor);
while(true){
// read message from keyboard
//place the tuple into the tuple space
lts.out(new Tuple(message_from_user));
// remove the message and discard the value returned
lts.in(message_from_user);
}
```

Each user creates a tuple space named "Chat_Room." Then it adds a weak reaction to this tuple space. This reaction has a template (*messageTemplate*) that will be compared against tuples in tuple space and, if any match is found, the reaction is fired. The parameter "reactor" is a reference to an object that implements a special method which will be called if the reaction is triggered. This method will receive a copy of the tuple and may send the message to the GUI. Here is a simple version of such a method:

```
public void reactsTo(Tuple t)
{print(extract(t));}
```

3 Security Extensions

In this section we revisit LIME by examining a set of extensions required to accomplish a smooth transition to a secure version of the model and its associated middleware.

Password Protected Tuple Spaces. Returning to the chat room application, it is easy to see that anyone having a PDA, even if she/he is unaware of the name given to the shared tuple space, can employ polymorphic matching over `LimeSystemTupleSpace` to return all the information needed to create a tuple space having the right name. One way to protect against such attacks is to require a password to be associated with each secure tuple space:

```
SecureLimeTupleSpace slts = new SecureLimeTupleSpace("name", "password");
```

An agent will be considered authorized if it has knowledge of both the tuple name and its password. An entry in `LimeSystemTupleSpace` corresponding to this tuple space will still exist but will not be recognizable as the password is used to generate the key that encrypts the actual name. Interestingly enough this will not permit for an agent to simply

read the name from `LimeSystemTupleSpace` and create its own local tuple. As we will see in the implementation section, the name of a tuple space suffers some transformations on the way from the user to `LimeSystemTupleSpace`. These changes will prevent an intruder from attempting to create an unprotected tuple space by copying the encrypted name of a tuple space from `LimeSystemTupleSpace` and not by generating it using the correct clear name and password.

Secure Communication. If a tuple space operation involves a remote execution on some other host whose agent contributes to the federated tuple space, the request will be sent across the wireless link and the results will be sent back over insecure wired or wireless lines. Eavesdropping is made easy by the fact that information travelling across the network consists of clear serialized Java objects. Secure communication between hosts is achieved by encrypting the messages associated with a given tuple space using the password supplied when the tuple space was created first (if any). The remote party is supposed to have access to the same password since sharing of the tuple space is taking place. For tuple spaces which are not protected, the messages will not be encrypted and the other party will need to know only the communication protocol in order to be able to deserialize the objects received in the request.

Tuple level access control. Even if we can now protect an entire tuple space, restrictions at the tuple level are still desirable in many applications.The reasons are two fold. In case of a secure tuple space shared among cooperating agents, tuple level protection can protect inadvertent tuple removal or access. Similarly, in an open tuple space this feature affords some level of protection against malicious agents.

A tuple may have a password to protect the tuple from removal (hereafter called remove-password) and a different password that protects the tuple from reading (hereafter called read-password). If the tuple has a read-password, a **rd** operation will retrieve it if it provides the same read-password or a remove-password equal to tuple's read-password, assuming that the fields match. This is because an agent that has the password to remove a tuple is also entitled to read the tuple. If a tuple has a remove-password, an **in** operation will have to provide the same remove-password to match this tuple. If the tuple has no remove-password but has a read-password, an **in** operation will need to provide the latter password to remove the tuple (Figure 1).

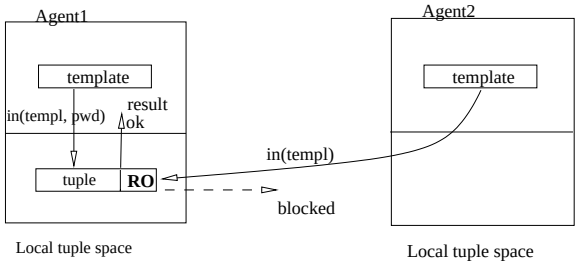

Figure 1: The execution of an **in** operation matching a Read-Only tuple. Agent1 is able to retrieve the tuple because it provides the (correct) remove-password, while Agent2 blocks because its template, even when it matches the data part of the tuple, does not satisfy the security requirements.

Group operations (e.g., **rdg**, **ing**, **outg**) as well as probe (i.e., nonblocking) operations (e.g., **rdp**, **inp**) behave similarly. Figure 2 shows how a **rdg** operation returns only the tuples that satisfy the security constraints, even if more tuples match the provided template.

As we will see in the implementation section, the passwords will be stored as special fields of a tuple with the matching policy set to exact value matching. For obvious reasons, no wildcards can be allowed in this fields' matching. It is also forbidden for an agent to push a protected tuple into some other's agent local tuple space. The new owner may not have the password to remove the tuple and will be stuck with it indefinitely.

In our chat application, if two agents want to exchange private information they need a secret key to protect the tuple space. While the authentication of the two is outside the scope of this example, we can show how they can establish a session key for their communication. Either one of the two agents can advertise its public key in a public tuple space. While everybody should be able to read this public key, the agent wouldn't like for anybody to be able to remove it,

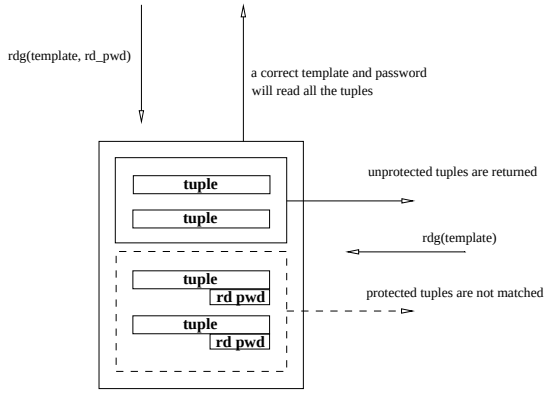

Figure 2: The execution of an **rdg** on a group of read-only and fully accessible tuples.

therefore will advertise it as read-only.

The advertisement of this public key can be done like this:

slts.out(publicKey, null, removePassword);

The *publicKey* parameter represents the tuple that contains the public key. The *null* parameter represents the lack of the read-password, which means that everybody is allowed to read it but the *removePassword* parameter indicates that only an agent that has this password can remove the tuple, named here "publicKey". Once the public key advertisement is secure, the two agents can agree on a private session key.

Discussion. Since full agent authentication requires a trusted computing base to certify identities in ad hoc settings, agents accessing the tuple space have to be authenticated on a different basis. Knowledge of an externally supplied password is one simple way of accomplishing this. Furthermore, passwords are user friendly, i.e., it is easier to handle passwords than keys.

Password distribution is an important issue but also problematic in ad hoc networks. We have to assume that the initial distribution is carried out external to the application. However, using the features provided by the model, the stage is set for password exchange between different agents. An agent (say Agent1) can advertise its public key in a read-only tuple (i.e., a tuple is protected with a remove-password, which is never given away, but no read-password). Another agent (say Agent2) can read this tuple and obtain Agent1's public key. The only problem Agent2 has to solve is to make sure that what it reads is indeed Agent1's public key and not a public key that is set up by an man-in-the-middle attack, which involves placing a fake key into Agent1's tuple space. This can be easily solved. All Agent1 has to do is to attempt to remove the tuple. If the tuple is read-only it must be the correct tuple. Agent2 reads the tuple from Agent1's local tuple space and, since it is a protected tuple, it couldn't have been planted there by another agent. Once Agent2 has Agent1's public key they can run a protocol to establish a secret session key. This secret key can be used to share password-protected tuple spaces or to exchange private information via password-protected tuples.

If a password is compromised, the only way to fix the problem is to remove the tuples protected by that password and rewrite them protected by a new password. If the password was protecting a tuple space, all the tuples have to be removed and rewritten in a new tuple space. Once a tuple space is created or a tuple is written to a tuple space, the password(s) protecting them cannot be changed anymore.

Any agent can notify the others if a password is compromised and should be changed. Each agent can register a strong reaction looking for 'password compromised' announcement tuples. When an agent wants to warn the others, all it has to do is write the warning tuple to the tuple space. Even if the attacker is now able to remove the tuple, the strong reactions will have to fire before the removal can complete. Thus, all interested agents can be notified. To resume collaboration, they will need to create another safe communication environment, i.e., to change the compromised password. There are several different ways this can be done. One would be to have each agent interact with an elected leader of the group. This leader could supervise the distribution of a new session key to all honest agents. This is a centralized approach (even though the leader is elected on the spot and not predefined) and a rather costly process of redistribution of a new session key (the leader will have to run a session key establishment protocol based on public key encryption with each other agent). A completely distributed approach would be to have each agent generate the new session key according to an algorithm known by all trusted agents. Thus they all generate the same new session key and are able to resume secure communication faster, as long as the key generation algorithms is not compromised as well.

Backward compatibility with older versions of LIME is insured by preserving the unprotected tuples and unprotected tuple spaces. The unprotected tuple spaces don't require encrypted communication and they fit the communication protocol of the tuple spaces from older versions of LIME.

4 Implementation

The security extensions introduced earlier were designed so as to have minimal impact over the programming interface offered to the developer. The original interface is still available. The extensions take password(s) as extra parameter(s) in the calls that handle protected targets (i.e., tuple space name and tuples). The secure inter host communication is automatically turned on by the usage of secure tuple spaces, therefore having no impact on the programmer interface.

For encryption we use a variant of the 3DES private key encryption algorithm that uses passwords instead of keys (the keys are generated internally from the provided passwords). We consider this algorithm secure enough for our purposes. The data being encrypted represents messages passed between hosts and not data that has to be stored safely. We also assume that Java language's protection mechanisms are robust enough not to allow incorrect access to internal data of an object (e.g., a *private* member of an object cannot be accessed by any other object). We do not address physical level attacks like wireless signal jamming.

4.1 Password Protected Tuple Spaces

The name of the tuple space is the key to gaining access to the information in that tuple space. To protect the information means to protect the name of the tuple space. `LimeSystemTupleSpace`, among other information, contains tuples that identify every tuple space (by name). Since the name is available in `LimeSystemTupleSpace`, the first step is to make the information obtained from `LimeSystemTupleSpace` unusable in its raw form. Changes are required to ensure that extracting the name of a protected tuple space from `LimeSystemTupleSpace` will no longer provide enough information for an agent to create a tuple space with the same name and share it with other agents thus gaining unauthorized access to its information.

To achieve this, some processing of the tuple space name will be done on the way from the constructor call, when creating the tuple space, to the internal storage of the name inside the system. The information available in `LimeSystemTupleSpace` will be the processed name of the tuple space. We make sure this information cannot be used in its form from `LimeSystemTupleSpace` and also that it cannot be generated incorrectly.

For this reason tuple spaces are split in two categories: tuple spaces that we want to protect and tuple spaces that are freely accessible, i.e., unprotected. If the user creates a tuple space that is intended to be secure, the user will have to provide a password. If no password is provided, the tuple space is assumed to be unprotected. For secure tuple spaces, the password is used to encrypt the name before marking it as a secure tuple space name and forwarding it to the previous implementation of LIME which will use it as if it were a regular string representing a name of a tuple space that will be used for sharing.

The interface the programmer uses to create secure tuple spaces is very similar to the interface offered by the previous version of LIME. The difference is that tuple spaces (secure or not) are created using the SecureLimeTupleSpace class. While the constructors still exist in their previous form, a new one was created, with an extra parameter: the password (Figure 3). If no password is provided, a simple, unprotected tuple space will be created, like in the previous version of LIME.

SecureLimeTupleSpace(java.lang.String name, java.lang.String password)
— creates a new secure tuple space using the public tuple space name and the password. This call places an entry in the **SecurityTable** mapping the encrypted name to the password.

Figure 3: The Call that Creates a Secure Tuple Space

The constructor call is the only place where the agent explicitly uses the password. Once the agent has the handle to the tuple space, it does not need the password anymore. The tuple space handle will enable the agent to access the tuple space for as long as the agent has it without having to provide the password. All methods will be invoked as before and will use the tuple space protection password transparently to the agent, if needed. A tuple space operation can only be called by the LIME agent that created it. When an operation is called on a tuple space, LIME verifies that it was called by the thread representing the agent that created it. Even if the handle of a tuple space is obtained correctly by an agent, it cannot be transferred and used by another agent. This is why it is not necessary to ask for the password when a tuple space operation is called.

The name of the secure tuple space is obtained from the provided name and password. This encrypted name appears in the `LimeSystemTupleSpace`. The tuple space name (encrypted name when a password is provided or the plain clear name if the tuple space is not meant to be protected) will be prefixed by a differentiator: letter "U" for unencrypted or "S" for secure tuple space. The tuple space called "blue" is different from the tuple space called "blue" and protected with password "pwd" (the latter will actually have the internal name $K_{pwd}(blue)$). They can coexist but no sharing takes place. The prefixes ensure that a tuple space cannot be created incorrectly. Since

they are internally added, they cannot be manipulated by agents. Reading the name of a (secure) tuple space from `LimeSystemTupleSpace` will not be enough to create an insecure tuple space with the same name. A prefix will be attached in front of whatever the programmer provides as a tuple space name. If an attacker reads the name of a protected tuple space from `LimeSystemTupleSpace` and tries to create a tuple space with the same name, there are two ways she/he could follow. One is to create the tuple space as an unprotected tuple space. In this case the system will add the "U" prefix and will not be shared with the original tuple space. The second attempt would be to trick the system to add the "S" prefix. To do so it will be necessary to create a secure tuple space. In this case the information retrieved by the attacker from `LimeSystemTupleSpace` is useless since she/he will need to provide the clear name and the correct password. There are no "blank" passwords that can be used to encrypt a text and to yield the same text as result. The prefixes also address the case when the result of encrypting the clear name of a tuple space coincides with the name of an unencrypted tuple space (before adding prefixes).

Using an old version of LIME (i.e., without the security features) on a different host will allow to illegally create the tuple space but no interaction takes place since all the communication with respect to a protected tuple space is encrypted.

The encrypted name of a protected tuple space and the password that protects it are important not only when the tuple space is created and shared but also later in inter-host communication. This is why the LIME server has a **SecurityTable** that stores entries of the form [encrypted name, password]. An entry is added to this table every time a new secure tuple space is created. When an operation is executed on the tuple space, if it runs on the local host of the issuer (identifiable by location parameters that define the projection of the tuple space) no further verification is needed. For executions of tuple space operations that span beyond the limits of issuer's host, the table will be used for more verifications. See Section 4.3 for details.

This **SecurityTable** is a very important target that has to be protected. Currently, only the default Java object protection mechanisms protect this table. We could encrypt it and provide a somewhat more difficult access to it but this would only shift the problem to protecting this password. Since this paper does not address the Java security model, we assume this model is secure enough for our research.

4.2 Tuple Level Protection.

To implement read-only tuples, several changes were needed to the previous version of LIME and to Lights, the tuple space implementation that LIME uses. Tuples are created in the same way as before. However, every tuple space operation will add to the end of the user specified fields (if any) three fields. They are in order: the read-password, the remove-password and the name of the operation that uses that tuple or template (e.g., "rd" for any type of read operation, "in" for any type of remove operation and "out" for any type of write operation). If either password is absent the field contains an instance of a **NULL** class (created to stand for the Java null but is a serializable object). The call without password parameters is equivalent to a call with both password parameters equal to **NULL**. When a tuple is written to the tuple space, the **out** method can specify both the read-password and the remove-password to protect the tuple in the tuple space.

To read a tuple, we have provided a **rd** method which takes a read-password beside the usual template. This method will construct a template that contains the **NULL** in remove-password's position and the read-password in the right place. For removing a tuple, the situation is similar. The **in** operation takes an extra parameter, the remove-password. The read-password is filled in with the same value since we consider that a template that is allowed to remove a tuple should also be allowed to read the tuple. In some cases one of the two passwords expands in the other's field from a semantic point of view. For example, if a tuple has a read password but no remove-password, a template trying to remove the tuple will need to have the read-password. Likewise, if a tuple has a read-password and a remove-password, and the template provides the remove-password for a read operation, access will be granted. Group operations are implemented similarly. An **outg** protects each tuple written to the tuple space with the password(s) provided (if any). The **ing** and **rdg** operations return only the tuples that satisfy the matching criteria for both the data and security parts. Figure 4 shows examples of tuple space access methods, involving passwords.

Even though the matching of the fields is carried out internally by LIME, the password fields in particular showed that sometimes it is very useful to have the possibility to chose the matching policy specific to a particular field. This led us to add to LIME the ability to select among three matching rules on a field by field basis. First, a field in a tuple may require the template to provide the exact value of the field for a match to be declared (i.e., the template

lts.out(ITuple tuple, char[] readPwd, char[] removePwd)
— writes a tuple to the tuple space and protects it against reading and/or removing. Any combination of the two passwords is permitted.

lts.rd(ITuple template, char[] readPwd)
— reads a tuple from the tuple space if the tuple and the template match(and the correct password is provided).

lts.in(ITuple template, char[] removePwd)
— removes a tuple from the tuple space if the tuple and the template match (and the correct password is provided).

Figure 4: The tuple space interaction operations.

must have the correct actual, hereafter called EXACT_VALUE match). Second, the tuple may restrict only the type of the template field to be the exact type of it's own field. (i.e., the template's field may be a formal but it must match the tuple's field type exactly, hereafter called EXACT_TYPE). Finally, the least constrained type of matching is when a tuple's field allows a wildcard in the template's corresponding field. For example, the Java Object object is a wildcard that will always match under these circumstances. This type of matching takes advantage of Java's OO polymorphism and this is why we'll call this policy POLY_TYPE.

When fields are added to a tuple, the type of matching can be specified for each of them. Figure 5 shows how fields are added to tuples and how to specify the matching policy for each of them. Field.EXACT_VALUE, Field.EXACT_TYPE and Field.POLY_TYPE are predefined integer Field that identify the EXACT_VALUE, EXACT_TYPE, and POLY_TYPE matching policies. If no policy is explicitly specified, POLY_TYPE is the default policy considered. Taking advantage of these extensions, the tuple passwords are transformed into fields subject to the EXACT_VALUE policy and added at the end of the tuple when written to the tuple space.

Tuple t = new Tuple();
t.addActual(new Integer(1), Field.EXACT_VALUE)
.addActual(new String("WU"));
— creates a tuple and adds fields it. To match this tuple, a template will need to have an **EXACT_VALUE** on its first field (that is an actual of type Integer and value 1). Since the second field doesn't have any matching policy specified, the **POLY_TYPE** is assumed, i.e., any formal of type String (or a supertype) would match the tuple.

Figure 5: Adding Fields and Matching Policy to a Tuple

4.3 Communication Level Protection

Operations on the federated tuple space cross host boundaries. These entail host to host communication over insecure lines. When an agent executes an operation that spans beyond the limits of the current host, an interceptor catches it, analyzes the tuple space that the message refers to (the name of the tuple space is always present in the message that travels across hosts) and takes the appropriate action (the use of the interceptor pattern [3] is natural for this case, when we add security to a system that in its initial design did not address this issue). It also offers a great deal of flexibility with respect to the choice of encryption protocol. Figure 6 shows how interceptors secure the communication between two hosts.

The interceptor checks wether the tuple space name appearing in the outgoing message is present in the **SecurityTable**. If the message refers to an unprotected tuple space (it is not in the table), the interceptor lets it pass through unchanged. If the tuple space is a secure one, the interceptor will extract from the table the password that corresponds to that tuple space and will use it to encrypt the message. The interceptor creates a packet that contains the encrypted message and the encrypted name of the tuple space the message refers to and forwards this packet to the other involved host. On the recipient's side, actions happen symmetrically. Another interceptor catches the incoming message, looks up the encrypted name of the tuple space in the local **SecurityTable** and if found, uses the corresponding password to decrypt the message. The message is then forwarded to the `LimeServer`. If the target tuple space is not a secure one, the name will not be found in the **SecurityTable** and the message will be forwarded unchanged to the `LimeServer`. The returned results are handled in the same way.

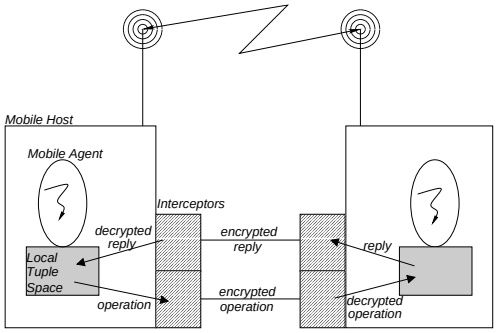

Figure 6: Interceptors catch messages and encrypt them before sending and then decrypt them upon receipt.

Special attention has to be paid when using password-protected tuples in unsecured tuple spaces. The traffic between two hosts is unprotected if it refers to an unsecured tuple space. If such a tuple space contains a password protected tuple (let's say the tuple has only a read-password) then a **rd** or **in** operation will need to provide the password along with the template. Let's assume a **rd** operation provides the correct password. Since the tuple space is not protected, the communication is not encrypted so the password travels in clear between the two hosts. A hacker could steal the password and use it then to remove the tuple (the tuple does not have a remove password so the read-password will be the only protection against removal as well). Password-protected tuples are safe to use in unprotected tuple spaces as long as the owner does not disclose the password (no message carrying a password should travel over insecure communication channels).

5 Wireless Dashboard Application

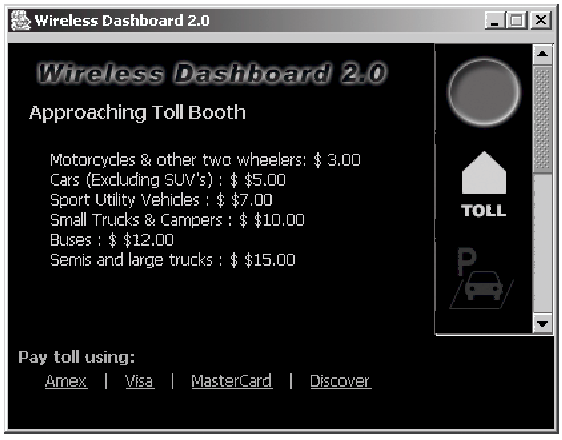

Figure 7: Automatic toll payment is only one of the features offered by a wireless dashboard application.

The extensions presented in this paper were first evaluated in a test application that allows a car driving down a highway to make an electronic payment to an approaching a tollbooth. As the car approaches the tollbooth, it discovers it, receives the list of prices, pays by credit card and continues its journey without stopping. Figure 7 shows a screen capture of the tollbooth application GUI.

The implementation is as follows. The car has an agent specialized in automatic payments (toll roads, parking, etc.). All these charge points are configured to establish contact with vehicle agents in a predefined, unprotected tuple space, called "payments". The agent in the car also has a tuple space called "payments". When the car approaches the tollbooth the two establish communication and tuple spaces merge based on the fact that they have the same name.

The agent in the car and the agent on the tollbooth will use the "payments" tuple space to establish a secret session key (noted SSK) for the purpose of collecting the payment from the car in a secure manner. The tollbooth advertises its public key (PK) in a read-only tuple in the unprotected "payments" tuple space (along with the list of prices for different car sizes, types of credit cards accepted, and other useful information) while keeping its pair (PK^{-1}) for itself (this will be used to decrypt incoming messages). The car reads the public key and generates a tuple that contains its identifier (license plate, or VIN number) and a secret session key, both encrypted with the tollbooth's public key: $< PK(name, SSK) >$.

Since the authentication part of the protocol is assumed (i.e., the car knows how to read a tuple **from** the tollbooth) the possible vulnerability is to read a tuple containing PK, planted there by an attacker. To protect

against this, the car agent will have to verify that the tuple is read-only (i.e., by failing in the attempt to remove it). The reader is reminded that, if the tuple is read-only, it could not have been placed there by anybody else since protected tuples cannot migrate (the API offered to the programmer does not allow writing protected tuples with an explicit destination; they will be placed, by default, in the producer's local tuple space). Another point of vulnerability is the public key encryption algorithm. In our implementation we used Bouncy Castle's implementation of RSA public key encryption algorithm. Since cryptography is outside the scope of this research, we also assume this encryption to be strong enough.

Using the name and *SSK* the car agent and the tollbooth agent will create a secure communication channel, a protected tuple space, accessible only to the two of them, where the payment will take place. After sharing the protected tuple space, the agent will send the credit card information to finish the payment, based on the selection made by the driver from the options advertised by the tollbooth. The transfer is done by having the tollbooth register a reaction for the payment tuple that the car will write to the protected tuple space. The tollbooth issues an electronic receipt in exchange. All sensitive information is handled internally to each agent and, when sent across platforms it travels encrypted. The tollbooth authenticates the car by accessing a trusted server through the wired infrastructure.

6 Related Work

In an open environment such as a computer network and especially in the presence of mobile code roaming across hosts, security is an important issue. Other projects also address this issue, trying to add different levels of protection to mobile agent systems and tuple space coordination of mobile agents. KLAIM (A Kernel Language for Agents Interaction and Mobility) [4] addresses the protection of data through the use of a capability based system combined with a type hierarchy based system for access control. In Secure Spaces [5] the authors employ a fine-grained approach to tuple matching mechanisms. They go down to field level to address security. They can protect each field individually by locking it with a password. This is somehow similar to using exact value matching for specific fields in the matching mechanisms described in this paper. Agents can be stopped from learning about data stored as tuples by requesting them to provide exact information in templates for tuple matching.

Several systems address the issue of protecting hosts from malicious agents. The D'Agents system [6] uses public key cryptography to authenticate incoming agents thus increasing the security of hosts. The more difficult problem of protecting the agent from curious hosts led to the approach of computing with encrypted functions [7], [8]. The key idea here is that mobile agents are able to decrypt code and data only if certain conditions are met by the computing environment or at a specific moment.

In [9] it is shown that strong typing is an essential concept for achieving strong security properties. The access rights are stored in a typed access rights matrix inspired by the HRU model[10]. A capability based system adapted to distributed computing is described in [11]. In Yalta [12] clients are logically grouped in dynamic coalitions. Yalta relies on certificates and certification authorities for emission, revocation and validation of certificates which leads to an architecture with several centralized hot points (certification authority and certification revocation service).

Distributed approaches to trust management are described in [13], [14], and [15]. They approach security issues in distributed computing using a centralized trusted entity to provide credentials that delegate permissions. These approaches are difficult to implement in ad hoc networks because in such environments it is almost impossible to maintain (or ensure access to) a centralized point of access to authorize credentials.

Administrative domains [16], [17] restrict the execution environment by logically dividing it into nested levels. The scope of a user's operations can be limited to his/her domain and the movement of running code can be restricted to well determined areas.

7 Conclusions

In this paper we presented a way to add security capabilities to the LIME coordination model. We chose LIME because it is the first coordination model designed to work in ad hoc networks. Our approach provides mechanisms needed to control *who* can do *what* and *how* with *which* tuples. We have showed that simple changes can transform a

coordination model into a platform suitable for the development of secure applications. The mechanisms are general and can solve real issues in terms of secure coordination in ad hoc networks.

Acknowledgements

This research was supported in part by the National Science Foundation under Grant No. CCR-9970939 and the Office of Naval Research under MURI Research Contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the research sponsors. The authors thank Rohan Sen for contributing to the development of the Wireless Dashboard application.

References

[1] Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems **7** (1985) 80–112

[2] Murphy, A., Picco, G., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proceedings of the 21^{st} International Conference on Distributed Computing Systems. (2001) 524–533

[3] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern Oriented Software Architecture. Volume 2. John Wiley & Sons, Ltd. (1999)

[4] R. De Nicola, Ferrari, G.L., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. Software Engineering **24** (1998) 315–330

[5] Vitek, J., Bryce, C., Oriol, M.: Coordinating agents with secure spaces. In: Proceedings of Coordination '99. LNCS, Springer Verlag (1999)

[6] Gray, R., Kotz, D., Cybenko, G., Rus, D.: D'Agents: Security in a multiple-language, mobile-agent system. In: Mobile Agents and Security. Volume 1419 of LNCS., Springer-Verlag (1998) 154–187

[7] Sander, T., Tschudin, C.F.: Protecting mobile agents against malicious hosts. In: Mobile Agent Security. LNCS, Springer-Verlag (1998) 44–60

[8] Riordan, J., B.Schneier: Environmental key generation towards clueless agents. In: Mobile Agents and Security. Volume 1419 of LNCS., Springer-Verlag (1998) 15–24

[9] Sandhu, R.S.: The typed access matrix model. In: Proceedings of the IEEE Symposium on Security and Privacy. (1992) 122–136

[10] Harrison, M., Ruzzo, W., Ullman, J.: Protection in operating systems. Communication of the ACM **19** (1976) 461–471

[11] Gong, L.: A secure identity-based capability system. In: IEEE Symposium on Security and Privacy. (1989) 56–65

[12] Byrd, G., Gong, F., Sargor, C., Smith, T.: Yalta: A secure collaborative space for dynamic coalitions. In: IEEE Workshop on Information Assurance and Security. (1989)

[13] Rivest, R.L., Lampson, B.: SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession (1996)

[14] Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., Ylonen., T.: Spki certificate theory. IETF, RFC 2693 (1999)

[15] Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems **10** (1992) 265–310

[16] Cardelli, L., Gordon, A.D.: Mobile ambients. In: Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS). Volume 1378., Springer-Verlag (1998) 140–155

[17] Vitek, J., Castagna, G.: Seal: A framework for secure mobile computations. In: ICCL Workshop: Internet Programming Languages. (1998) 47–77